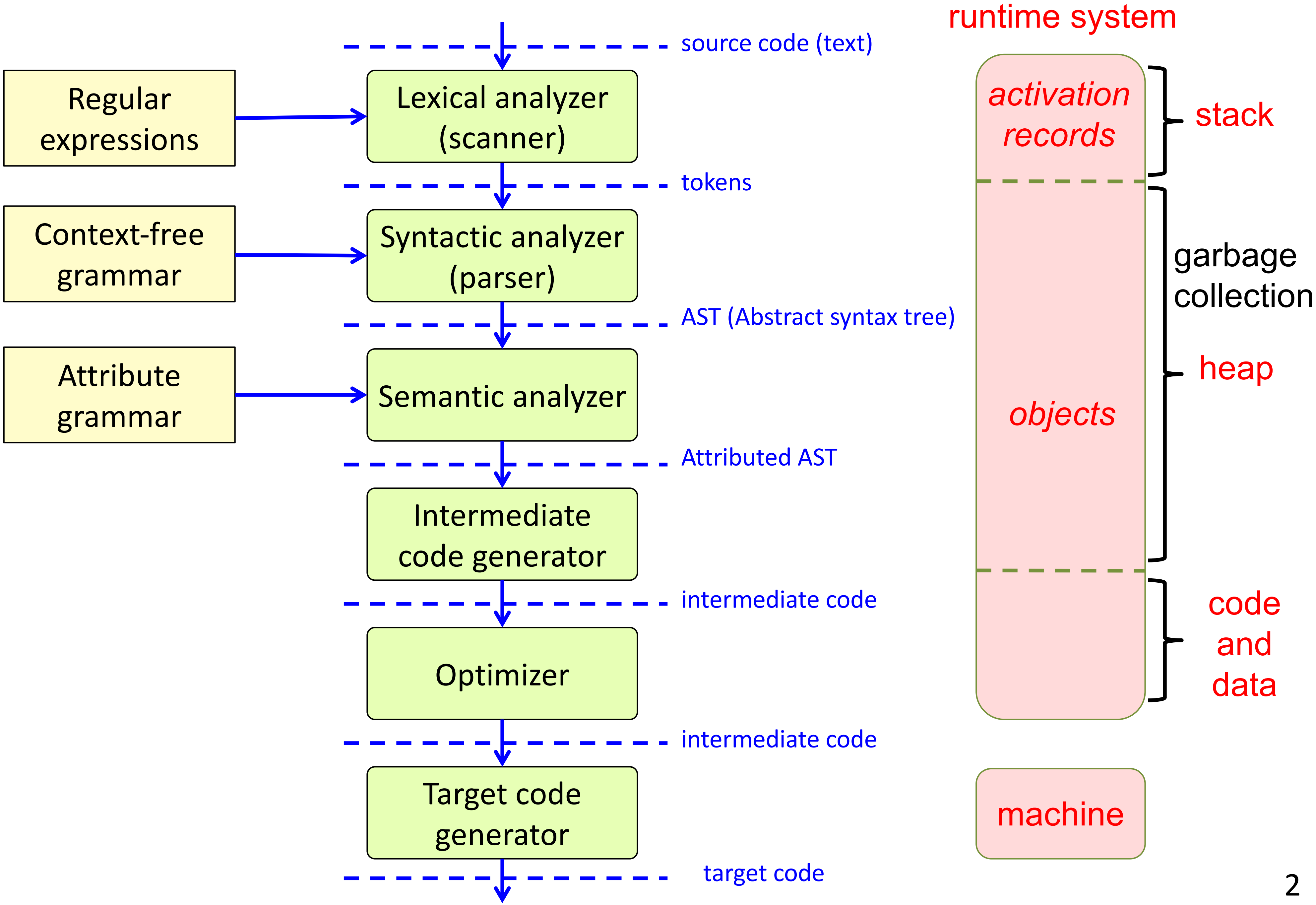
EDAN65: Compilers, Lecture 10

Runtime systems

Görel Hedin

Revised: 2023-09-25

This lecture



Runtime systems

Organization of data

- Global/static data
- Activation frames (method instances)
- Objects (class instances)

Method calls

- Call and return
- Parameter transmission

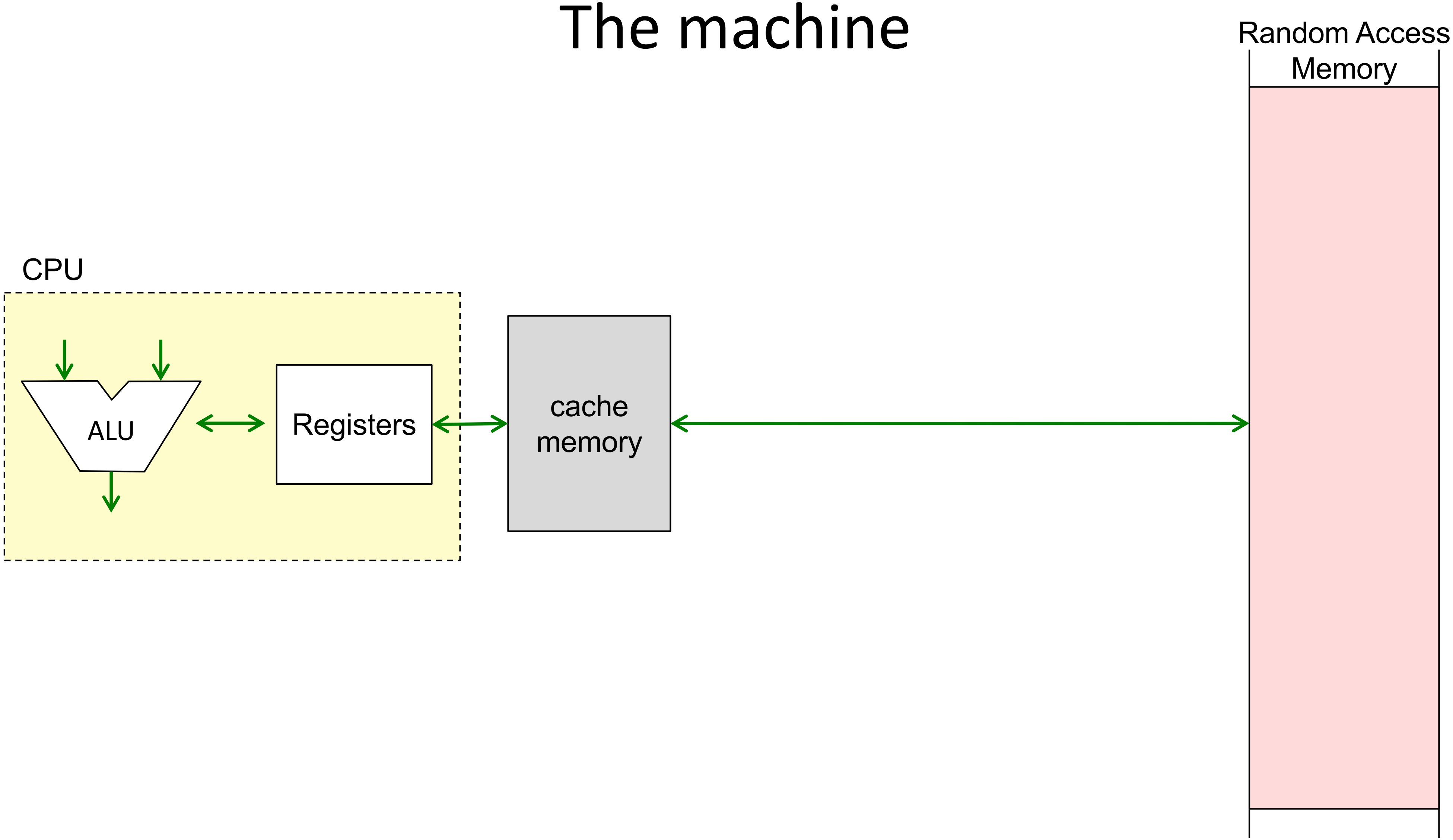
Access to variables

- Local variables
- Non-local variables

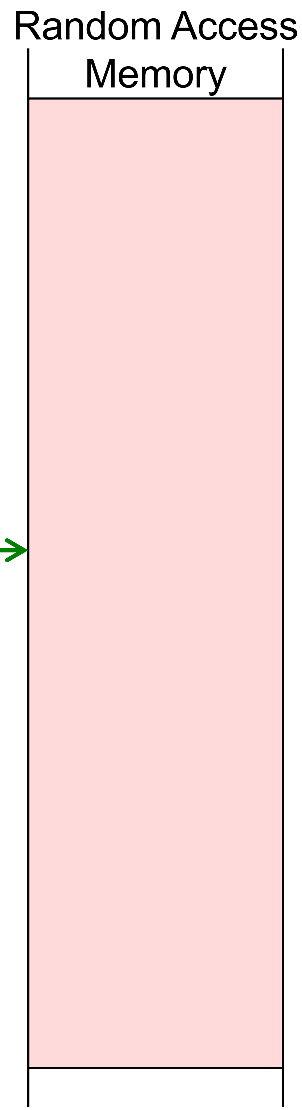
Object-oriented constructs

- Inheritance
- Overriding
- Dynamic dispatch
- Garbage collection

The machine

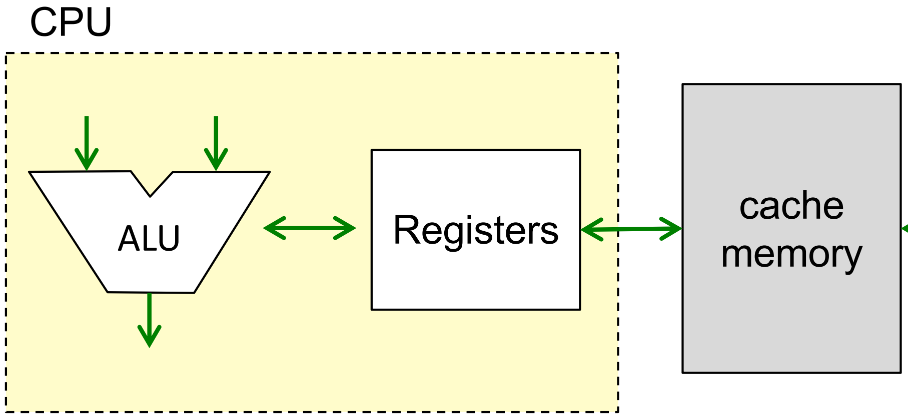


The machine



RAM:
Typically divided into memory *segments*

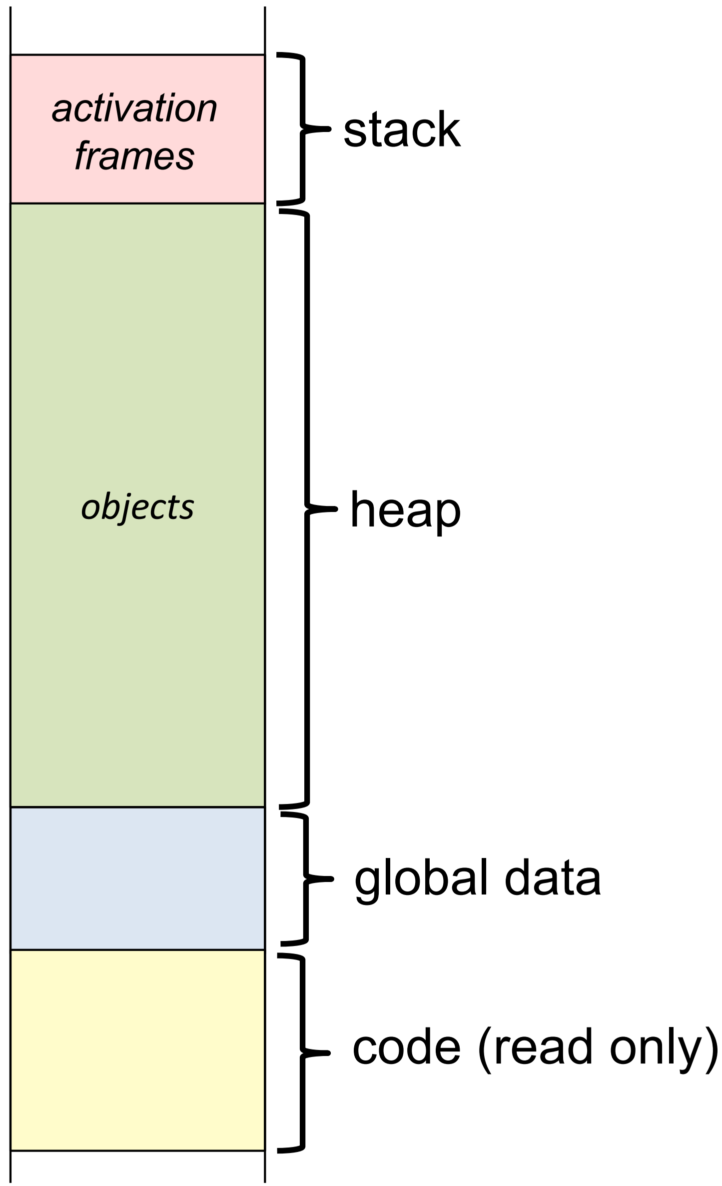
Typically byte-addressed.
Like a very large array.



Registers:
Some dedicated: program counter, stack pointer, ...
Some general purpose (for computations)
Typically 32 registers.

32-bit machine: Each register is 32 bits wide. Can address max 2^{32} bytes of RAM = 4GB.
64-bit machine: Each register is 64 bits wide. Could theoretically address max 2^{64} bytes of RAM (in practice, use perhaps 48 bits to address max 256 TB).

Example memory segments

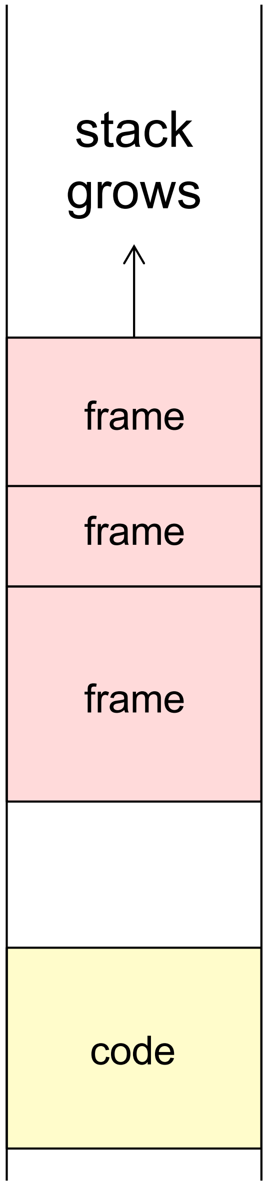


Stack of activation frames

The data for each method call is stored in an **activation frame**

Synonyms:
activation record
activation
stack frame
frame

Swedish:
aktiveringspost

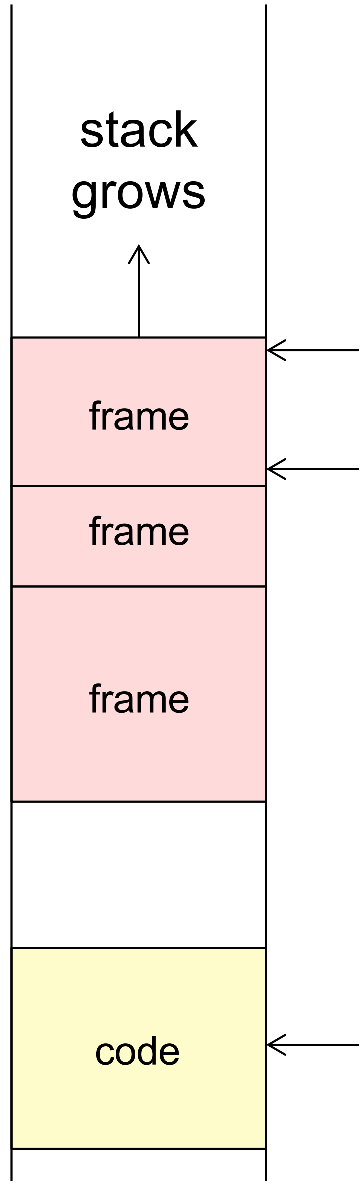


Stack of activation frames

The data for each method call is stored in an **activation frame**

Synonyms:
activation record
activation
stack frame
frame

Swedish:
aktiveringspost



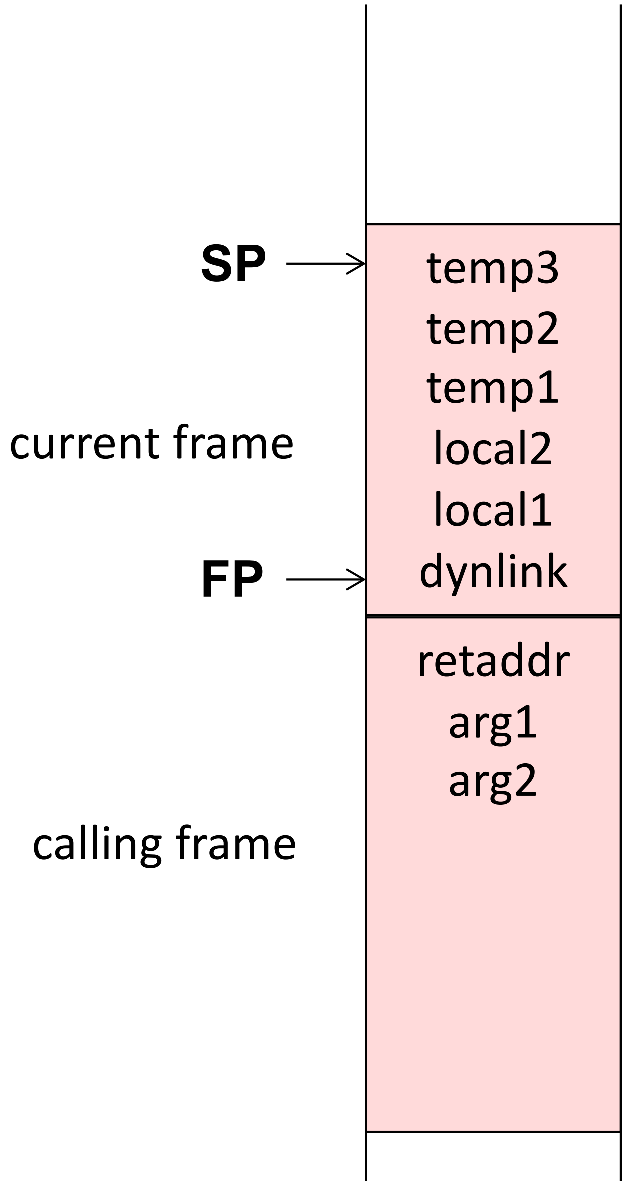
Some dedicated registers:

SP – Stack Pointer. The top of the stack.

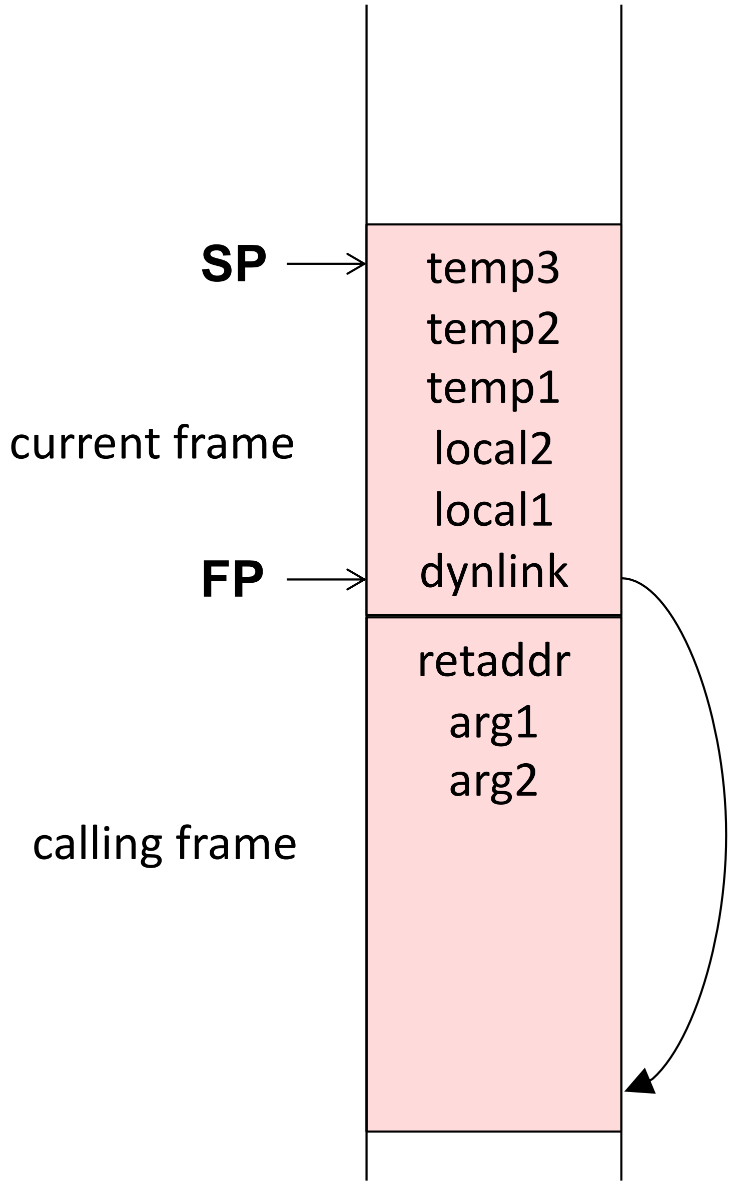
FP – Frame Pointer. The first word of the current frame

PC – Program counter. The currently executing instruction.

Example frame layout



Example frame layout



temps: Temporary variables

locals: Local variables

dynlink: Dynamic link – points to frame of calling method

retaddr: Saved PC - where to jump at return

args: Arguments to current frame.

The calling method pushes arguments on the stack.
The return value is placed in a register.

Frame pointer

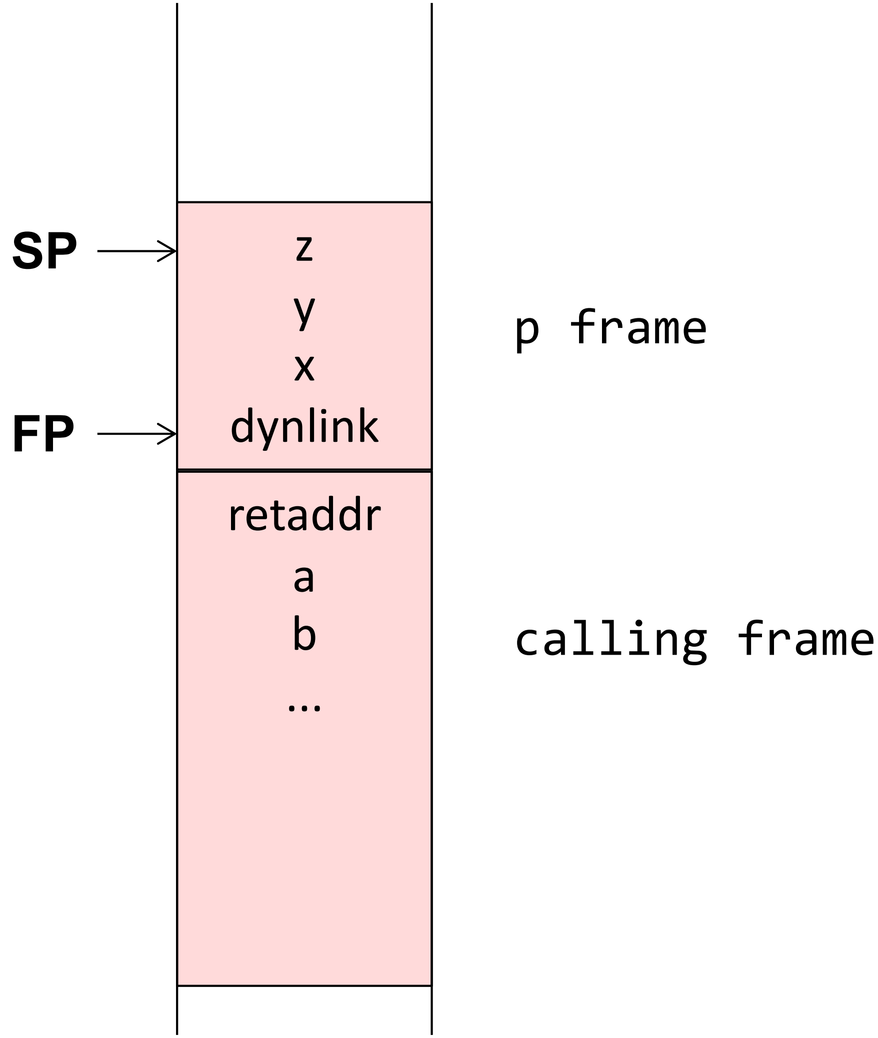
Used for accessing arguments and variables in the frame

```
void p(int a, int b) {
  int x = 1;
  int y = 2;
  int z = 3;
  ...
}
```

Frame pointer

Used for accessing arguments and variables in the frame

```
void p(int a, int b) {
  int x = 1;
  int y = 2;
  int z = 3;
  ...
}
```



Stack pointer

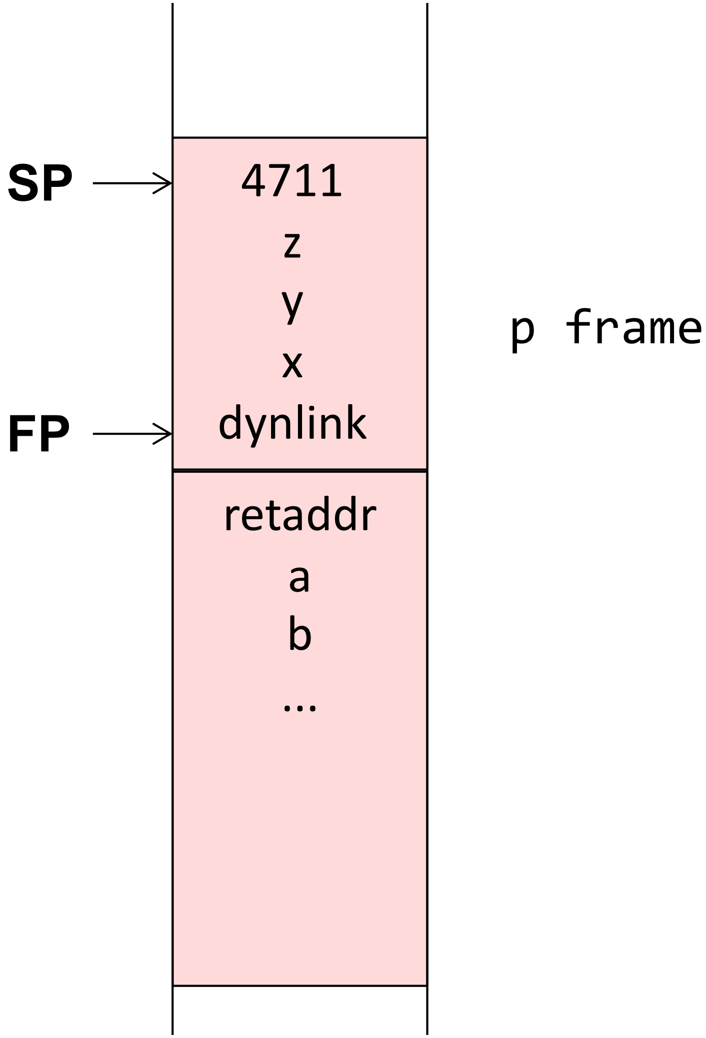
Used for growing the frame, e.g., at a method call

```
void p(int a, int b) {
  int x = 1;
  int y = 2;
  int z = 3;
  q(4711);
}
```

Stack pointer

Used for growing the frame, e.g., at a method call

```
void p(int a, int b) {
  int x = 1;
  int y = 2;
  int z = 3;
  q(4711);
}
```



The argument 4711 is pushed on the stack before calling q

Dynamic link

Points to the frame of the calling method

```
void p1() {
  int x = 1;
  int y = 2;
  p2();
}

void p2() {
  int z = 3;
  p3();
}

void p3(){
  int v = 4;
}
```

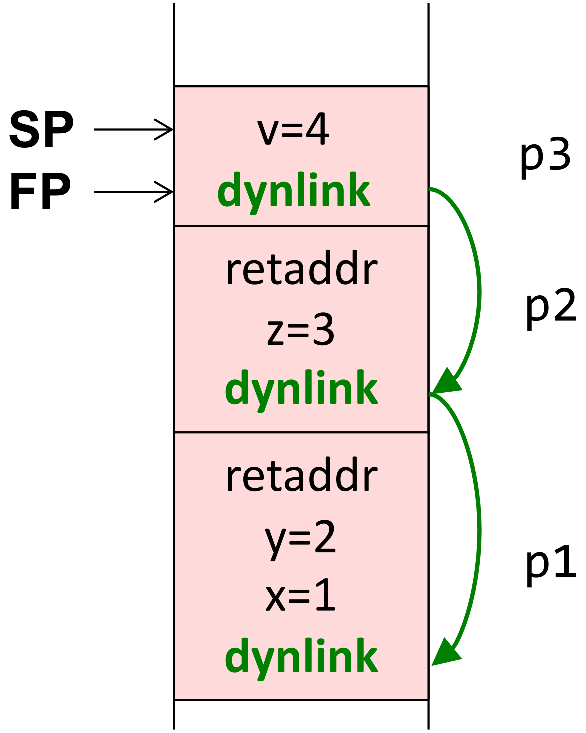
Dynamic link

Points to the frame of the calling method

```
void p1() {
  int x = 1;
  int y = 2;
  p2();
}

void p2() {
  int z = 3;
  p3();
}

void p3(){
  int v = 4;
}
```



Used for restoring FP when returning from a call.

Recursion

```
int f(int x) {
  bool ready = x <= 1;
  if (ready)
    return 1;
  else
    return x * f(x-1);
}
```

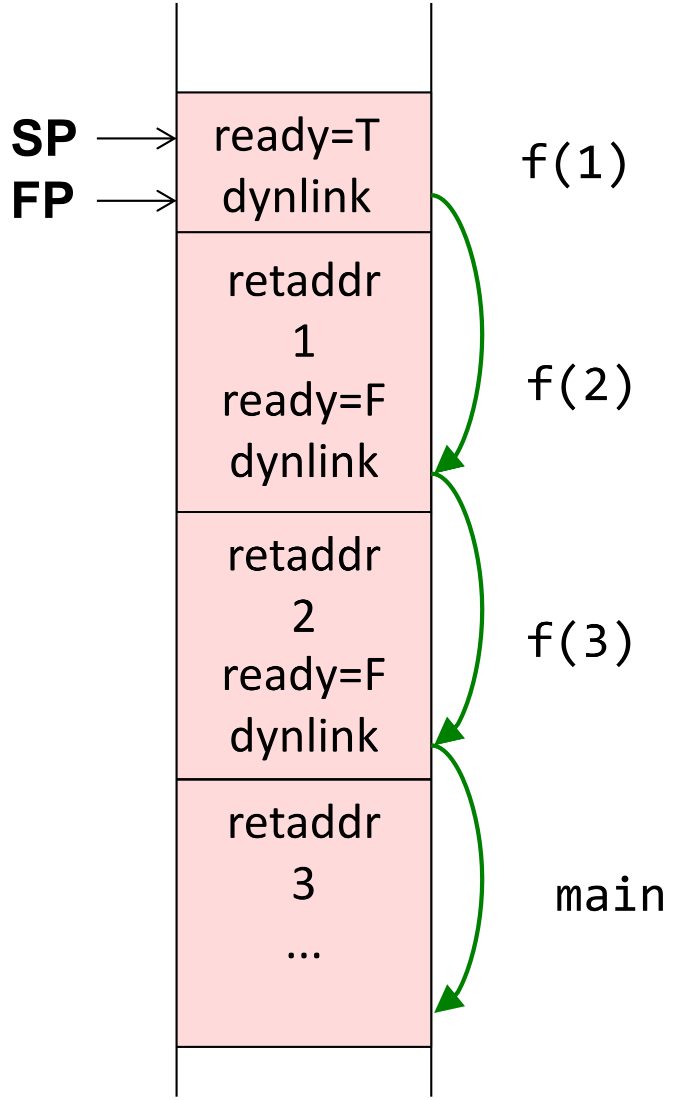
```
void main() {
  ...
  f(3);
  ...
}
```

Recursion

Several activations of the same method

```
int f(int x) {
  bool ready = x <= 1;
  if (ready)
    return 1;
  else
    return x * f(x-1);
}
```

```
void main() {
  ...
  f(3);
  ...
}
```



Nested methods

Static link – an implicit argument that points to the frame of the enclosing method.
Makes it possible to access variables in enclosing methods.

```
void p1() {
  int x = 1;
  int y = 2;

  void p2() {
    int z = y+1;
    p3();
  }

  void p3(){
    int t = x+3;
  }

  p2(); y++;
}
```

The methods are *nested.*
Supported in Algol, Pascal, Python,
but not in C, Java...

Nested methods

Static link – an implicit argument that points to the frame of the enclosing method.
Makes it possible to access variables in enclosing methods.

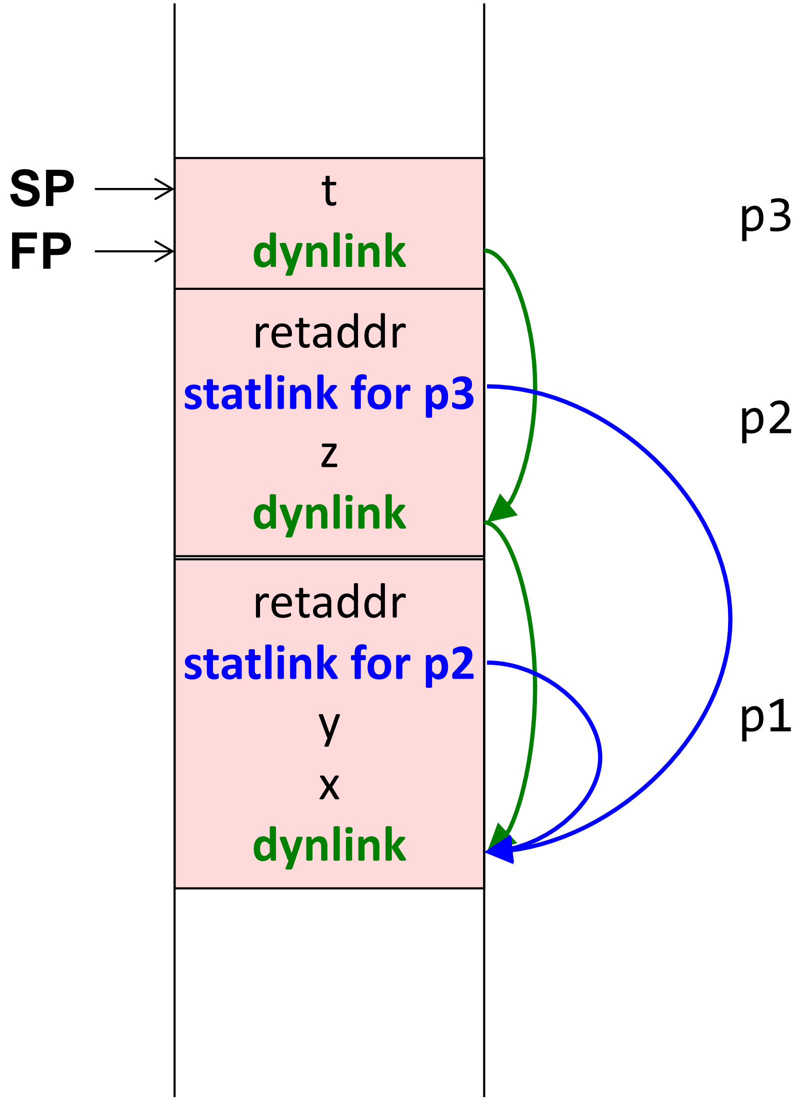
```
void p1() {
  int x = 1;
  int y = 2;

  void p2() {
    int z = y+1;
    p3();
  }

  void p3(){
    int t = x+3;
  }

  p2(); y++;
}
```

The methods are *nested*.
Supported in Algol, Pascal, Python,
but not in C, Java...



Objects and methods

This pointer – an implicit argument. Corresponds to the static link.
Makes it possible to access fields in the object.

```
class A {
  int x = 1;
  int y = 2;

  void ma() {
    x = 3;
  }
}
```

```
class B {
  void mb() {
    A a = ...;
    a.ma();
  }
}
```

```
void main() {
  new B().mb();
}
```

Objects and methods

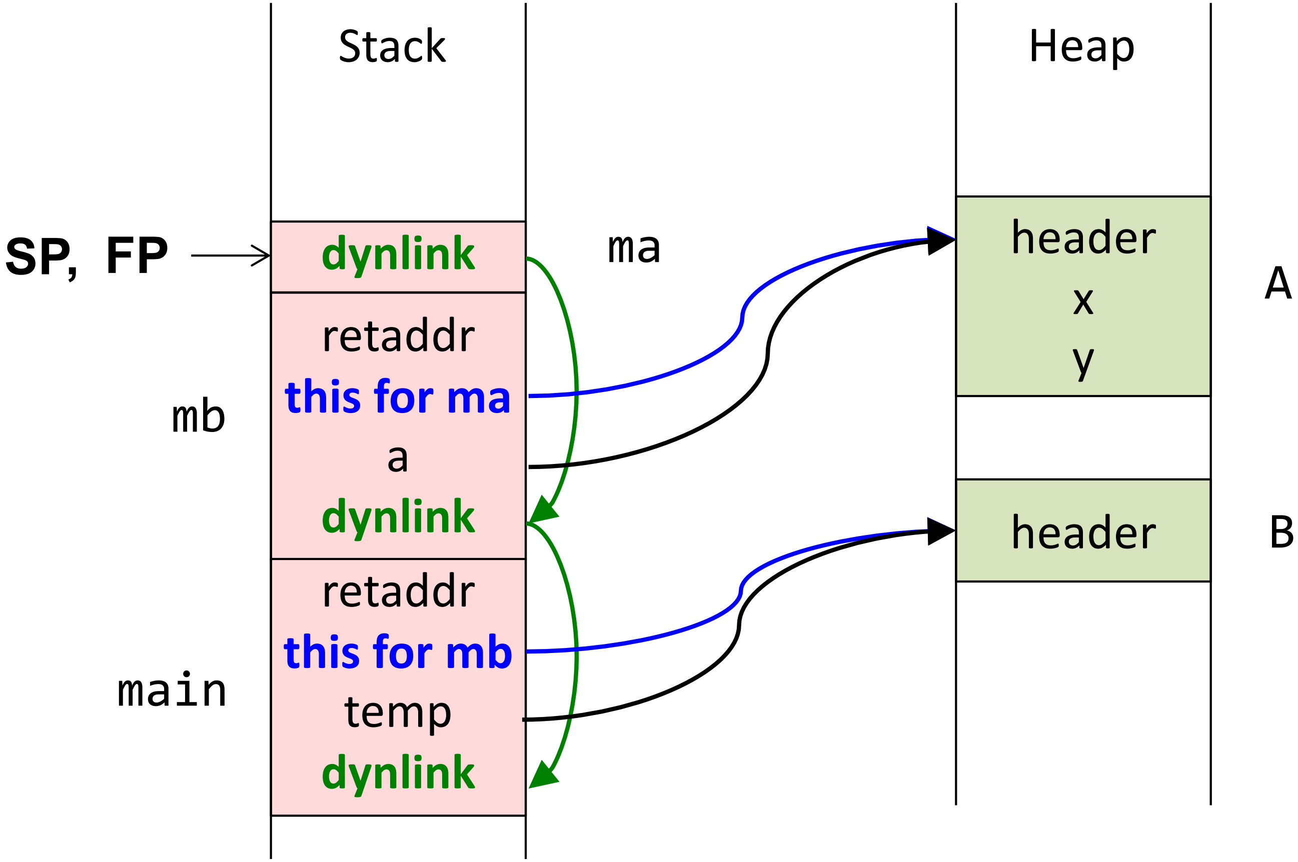
This pointer – an implicit argument. Corresponds to the static link.
Makes it possible to access fields in the object.

```
class A {
  int x = 1;
  int y = 2;

  void ma() {
    x = 3;
  }
}
```

```
class B {
  void mb() {
    A a = ...;
    a.ma();
  }
}
```

```
void main() {
  new B().mb();
}
```

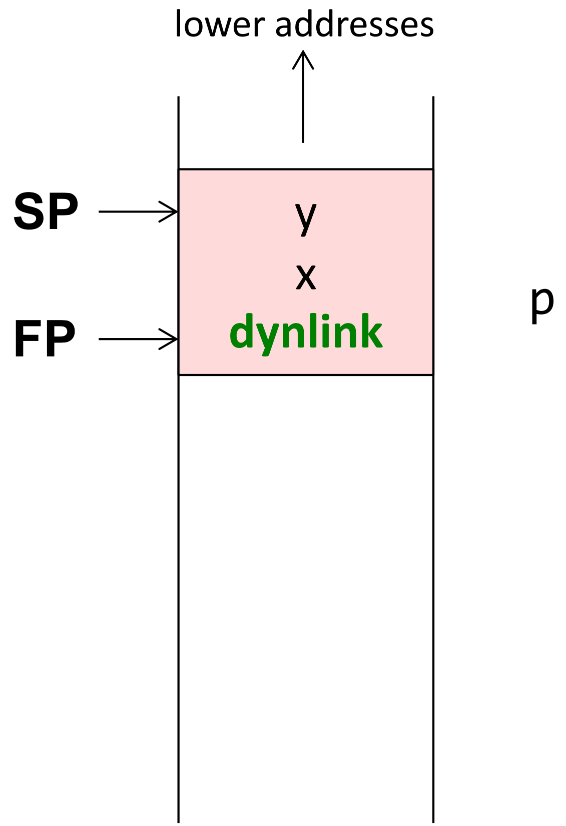


Access to local variable

```
void p() {
  int x = 1;
  int y = 2;
  y++;
  ...
}
```

Assume each word is 8 bytes.

The compiler computes addresses relative to FP



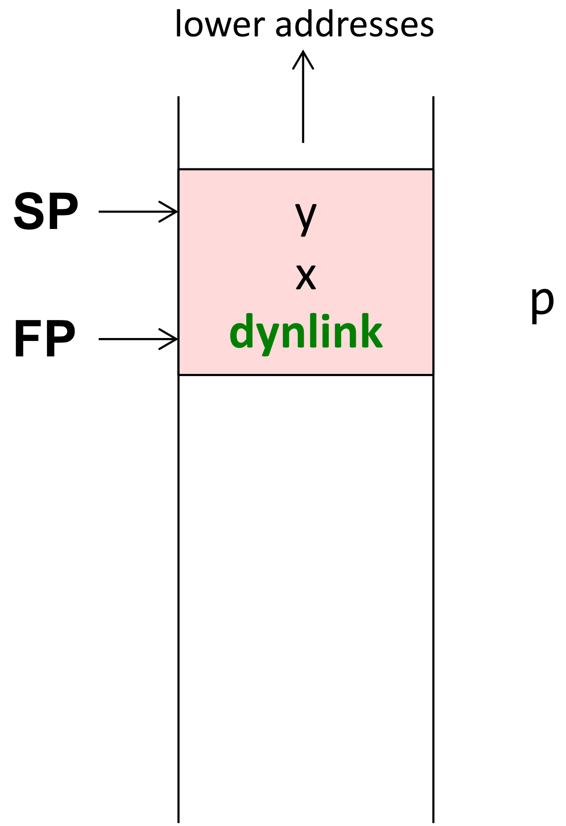
Access to local variable

```
void p() {
  int x = 1;
  int y = 2;
  y++;
  ...
}
```

Assume each word is 8 bytes.

The compiler computes addresses relative to FP:

var	offset	address
x	1	FP-8
y	2	FP-16



Typical assembly code for y++

```
SUB   FP  16  R1      // Compute address of y, place in R1
LOAD  R1  R2          // load value of y into R2
INC   R2              // increment R2
STORE R2  R1          // store new value into y
```

Computing offsets for variables

```
void p() {
  boolean f1 = true;
  int x = 1;
  boolean f2 = false;
  if (...) {
    int y = 2;
    ...
  }
  else {
    int z = 3;
    ...
  }
  ...
}
```

Simple solution: just number all the variables and place them in consecutive words.

Computing offsets for variables

```
void p() {
  boolean f1 = true;
  int x = 1;
  boolean f2 = false;
  if (...) {
    int y = 2;
    ...
  }
  else {
    int z = 3;
    ...
  }
  ...
}
```

Simple solution: just number all the variables and place them in consecutive words.

Possible optimizations:

- Variables with disjoint lifetimes can share the same memory cell
- Booleans can be stored in bytes or bits
- Variables can be reordered to make efficient use of space (e.g., aligning ints and floats to words)

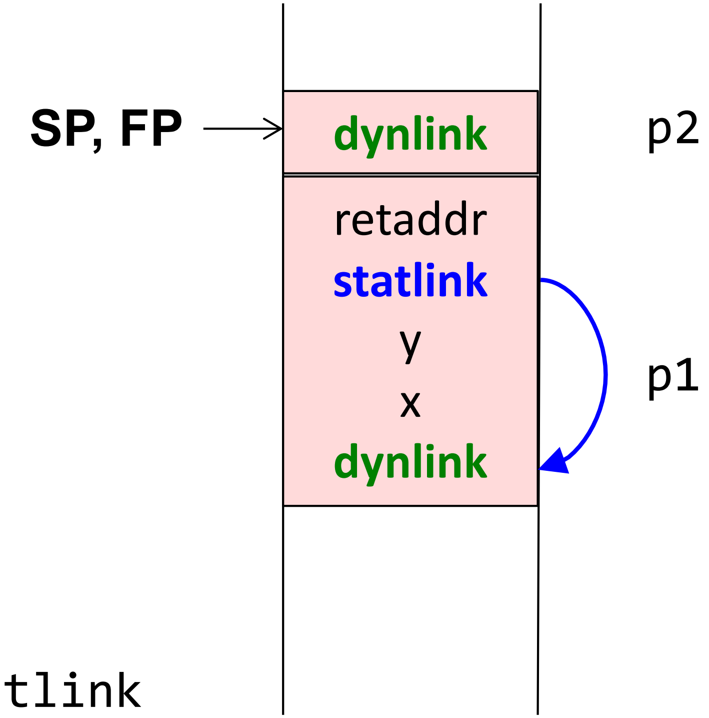
...

Access to non-local variable

```
void p1() {
  int x = 1;
  int y = 2;
  void p2() {
    x++;
  }
  p2();
}
```

Access to non-local variable

```
void p1() {
  int x = 1;
  int y = 2;
  void p2() {
    x++;
  }
  p2();
}
```



The compiler knows that x is available in an instance of p1 (the enclosing block).

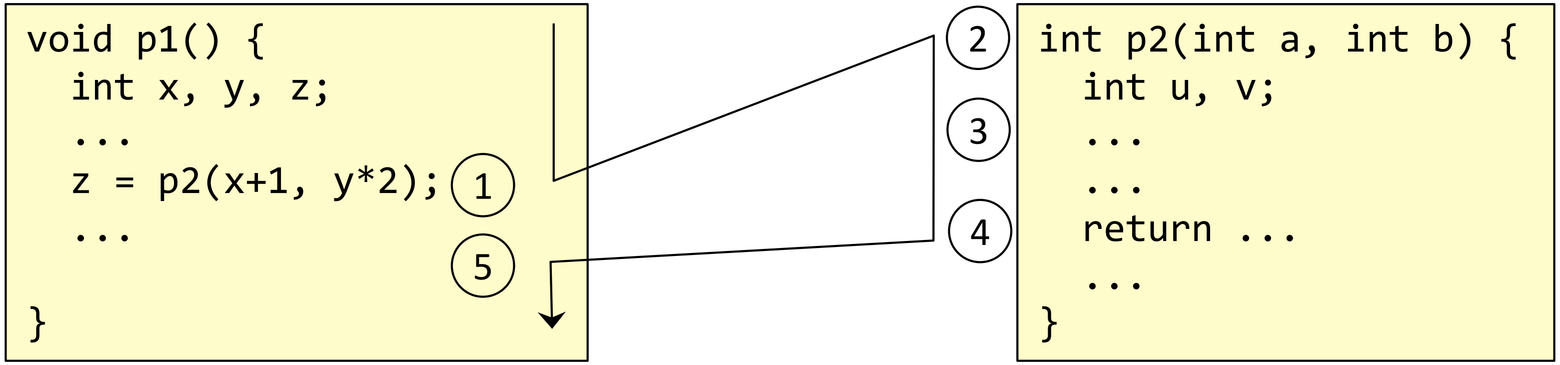
Follow the static link once to get to the enclosing frame

```
ADD   FP  16  R1  // Compute address of statlink
LOAD  R1  R2      // Get address to p1's frame

SUB   R2  8   R3  // Compute the address of x
LOAD  R3  R4      // Load x into R4
INC   R4          // Increment
STORE R4  R3      // Store the new value to memory
```

For deeper nesting, follow multiple static links.

Method call



1. Transfer arguments: Push them on the stack.
   Do call: Push the return address and jump to the called method (CALL instruction)

2. Allocate new frame: Push FP (dynamic link) and move FP.
   Allocate space for local vars: Move SP.

3. Run the code for p2.

4. Save the return value in a dedicated register, e.g. RAX on Intel.
   Deallocate local vars: Move SP back.
   Deallocate the frame: Move FP back. Pop FP.
   Do return: Pop return address and jump to it (RET instruction)

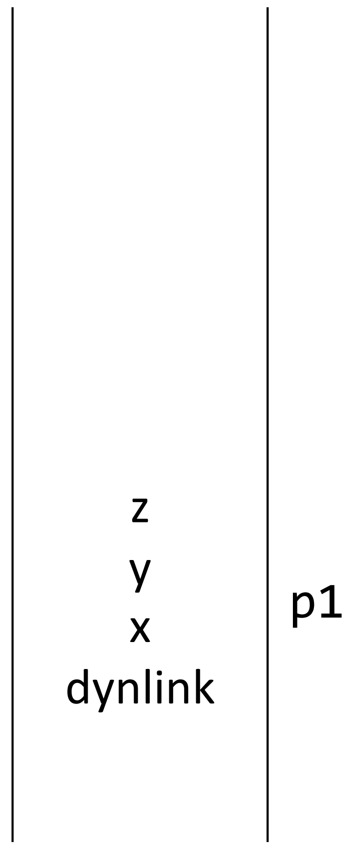
5. Pop arguments. Continue executing in p1.

Method call

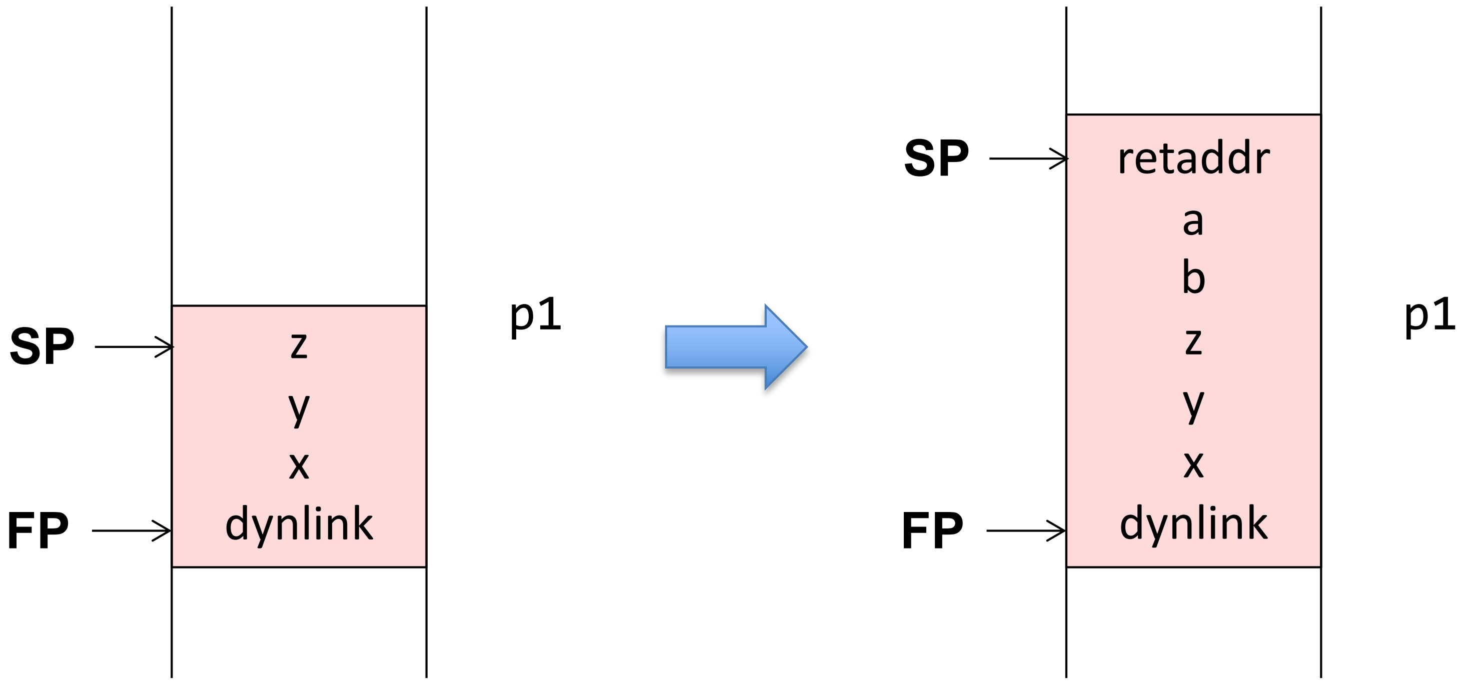
```
void p1() {
  int x, y, z;
  ...
  z = p2(x+1, y*2);
  ...

}
```

```
int p2(int a, int b) {
  int u, v;
  ...
  ...
  return ...
  ...
}
```



Step 1: Transfer arguments and call.

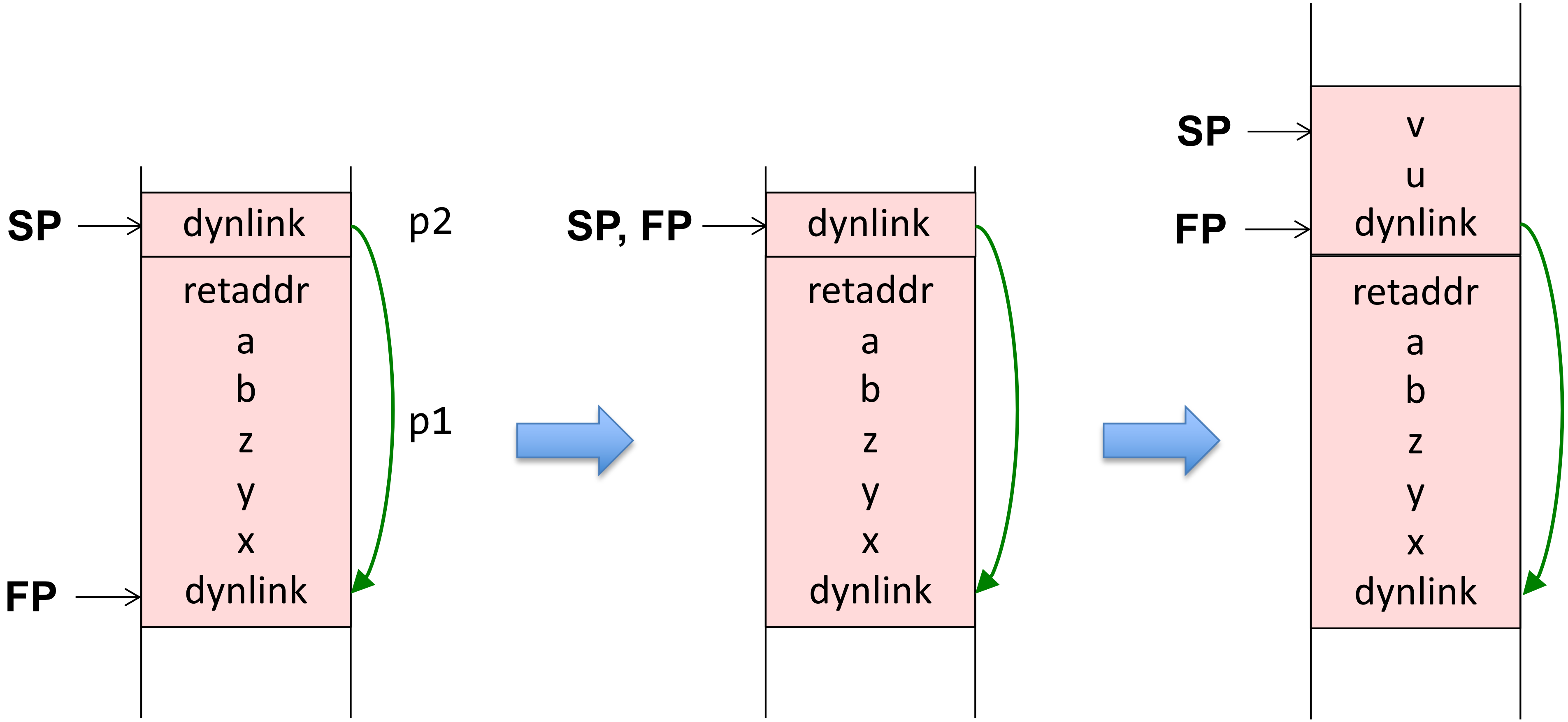


Transfer arguments:
- Push the arguments on the stack

Do the call:
- Compute the return address (e.g., PC+16) and push it on the stack.
- Jump to the code for p2.

(An instruction "CALL p2" accomplishes these two things.)

Step 2: Allocate the new frame

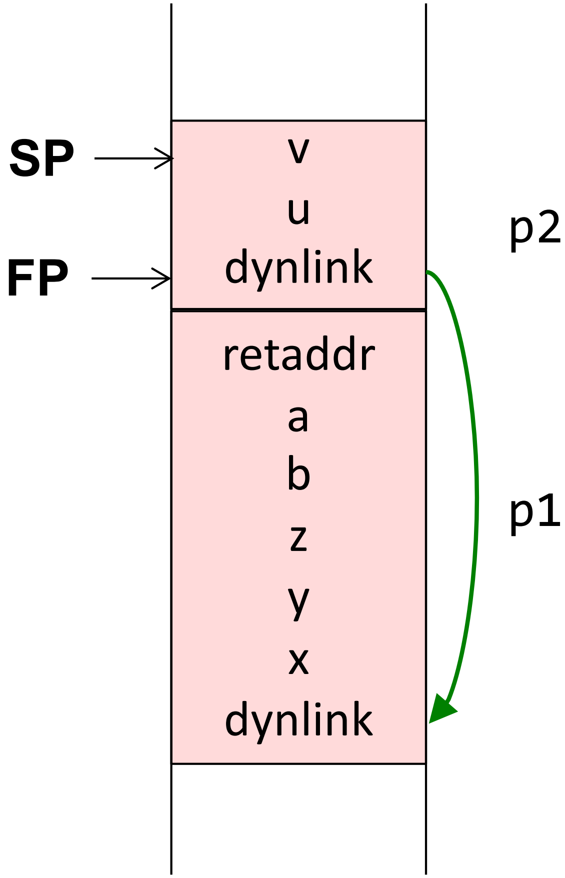


push the dynamic link (current FP)

set FP to the new frame

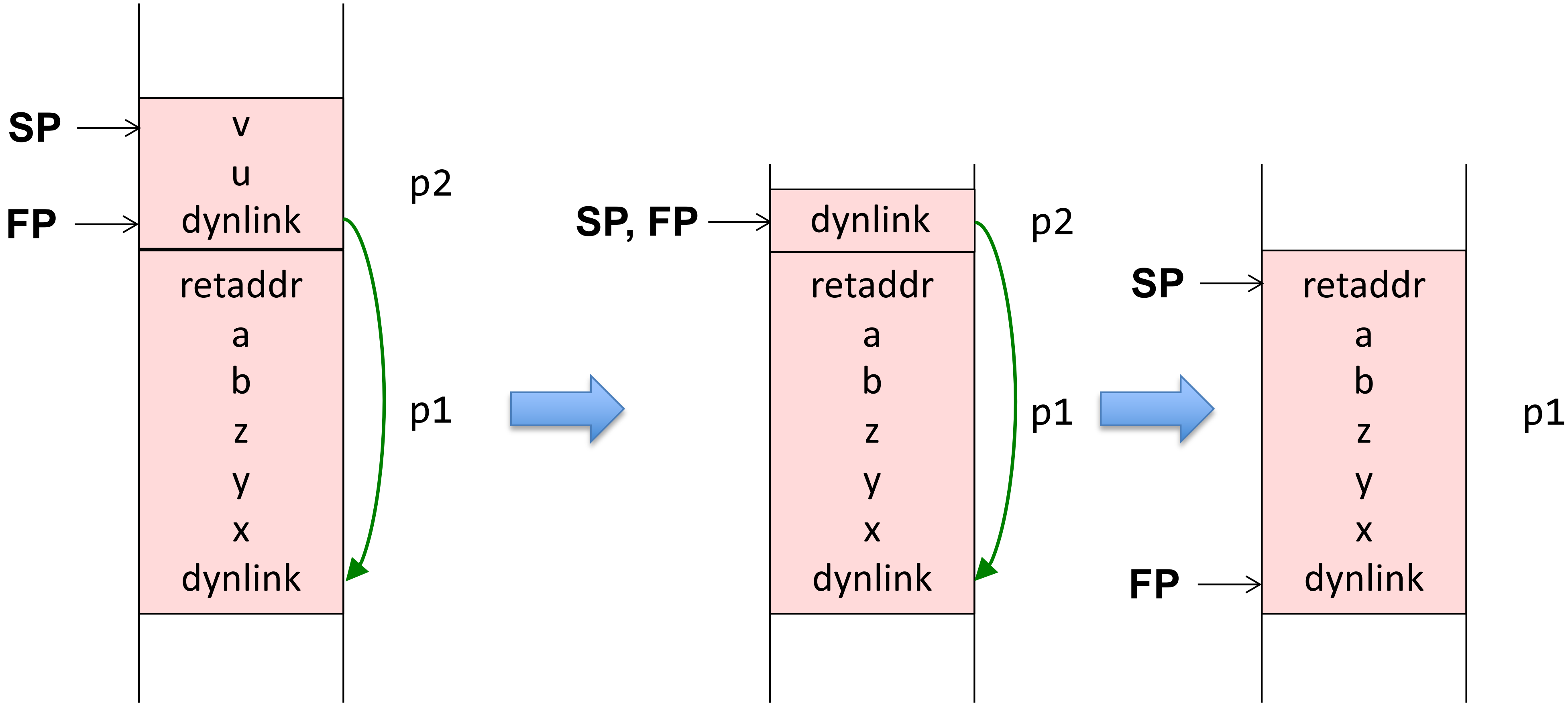
move SP to allocate space for new locals

Step 3: Run the code for p2



run the code for p2

Step 4: Deallocate and return



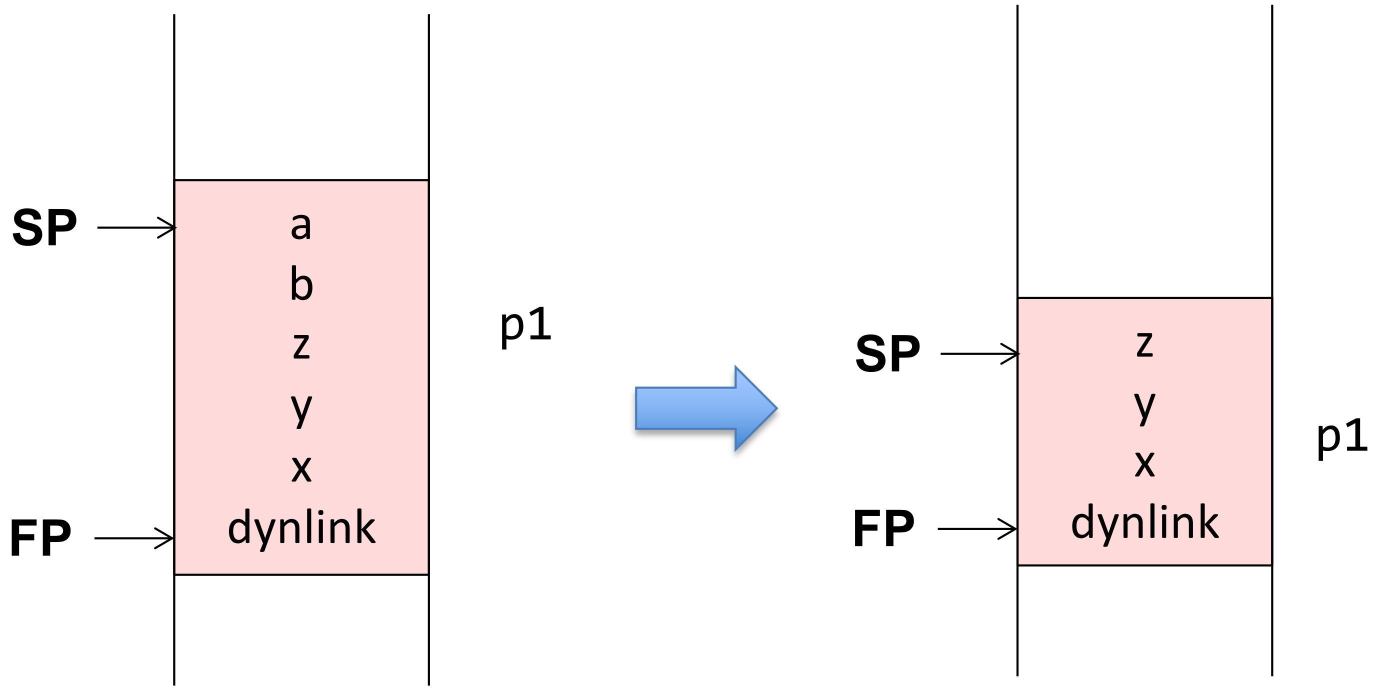
Store the return value in a register.

Deallocate locals: Move SP back to FP.

Pop FP and set FP to old value (dynlink)

Then do the RET instruction
(will pop the return address and jump to it)

Step 5: Continue executing in p1



- Pop the arguments
- Continue executing code in p1

What the compiler needs to compute

For uses of locals and arguments

- The offsets to use (relative to the Frame Pointer)

For methods

- The space needed for local declarations and temporaries.
  (Typically use push/pop for allocation/deallocation of temps.)

If nested methods are supported

- The number of static levels to use for variable accesses (0 for local vars)
- The number of static levels to use for method calls (0 for local methods)

Calling conventions

(agreements between caller and callee)

Calling conventions
(agreements between caller and callee)

Examples:

Conventions for the stack, e.g.:

- What direction the stack grows, e.g. from higher to lower addresses.
- What registers are used for stack and framepointer, e.g., RSP and RBP on Intel architecture

How arguments and return values are transferred, e.g.:

- Arguments passed on the stack frame in backwards order
- Return value passed in dedicated register, e.g., RAX on Intel architecture
- How implicit arguments like static link and "this" pointer are passed.

Caller-save registers: Registers that the caller must save before calling.

Callee-save registers: Registers that the callee (the called method) must save before using them. The values of the saved registers must be restored before return.

Summary questions

- What is the difference between registers and memory?
- What typical segments of memory are used?
- What is an activation frame?
- Why are activation frames put on a stack?
- What are FP, SP, and PC?
- What is the static link? Is it always needed?
- What is the dynamic link?
- What is meant by the return address?
- How can local variables be accessed?
- How can non-local variables be accessed?
- How does the compiler compute offsets for variables?
- What happens at a method call?
- What information does the compiler need to compute in order to generate code for accessing variables? For a method call?
- What is meant by ”calling conventions”?